Report on the Algorithmic Language ALGOL 60

Peter Naur (*Editor*)

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger

Dedicated to the Memory of William Turanski

INTRODUCTION

Background

After the publication of a preliminary report on the algorithmic language Algol,[1,2] as prepared at a conference in Zürich in 1958, much interest in the Algol language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an Algol implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL *Bulletin*, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the Algol language and preparing a final report. At a European Algol Conference in Paris in November 1959 which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represent the following organizations: Association Francaise de Calcul, British Computer Society, Gesellschaft fur Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to Algol was requested to send his comments to the ACM *Communications* where they were published. These comments then became the basis of consideration for changes in the Algol language. Both the Share and USE organizations established Algol working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on Algol that had been sent to the ACM *Communications*. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

January 1960 Conference

The thirteen representatives,[3] from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960.

Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

As with the preliminary Algol report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

Reference Language

1. It is the working language of the committee.
2. It is the defining language.

[1] Preliminary report—International Algebraic Language, *Comm. Assoc. Comp. Mach. 1*, No. 12 (1958), 8.

[2] Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages and the GAMM Committee on Programming, edited by A. J. Perlis and K. Samelson, *Numerische Mathematik* Bd. *1*, S. 41–60 (1959).

[3] William Turanski of the American group was killed by an automobile just prior to the January 1960 Conference.

May 1960 vol 3 no 5

3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.

4. It is the basic reference and guide for compiler builders.

5. It is the guide for all hardware representations.

6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.

7. The main publications of the ALGOL language itself will use the reference representation.

PUBLICATION LANGUAGE

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g., subscripts, spaces, exponents, Greek letters).

2. It is used for stating and communicating processes.

3. The characters to be used may be different in different countries, but univocal correspondence with reference representation must be secured.

HARDWARE REPRESENTATIONS

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.

2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.

3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

For transliteration between the reference language and a language suitable for publications, among others, the following rules are recommended.

Reference Language	*Publication Language*
Subscript bracket [ ]	Lowering of the line between the brackets and removal of the brackets
Exponentation ↑	Raising of the exponent
Parentheses ( )	Any form of parentheses, brackets, braces
Basis of ten $_{10}$	Raising of the ten and of the following integral number, inserting of the intended multiplication sign

DESCRIPTION OF THE REFERENCE LANGUAGE

> Was sich überhaupt sagen lässt, lässt sich klar sagen; und wovon man nicht reden kann, darüber muss man schweigen.
> LUDWIG WITTGENSTEIN.

1. Structure of the Language

As stated in the introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication—and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called assignment statements.

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe, e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refer to another, statements may be provided with labels. Sequences of statements may be combined into compound statements by insertion of statement brackets.

Statements are supported by declarations which are not themselves computing instructions, but inform the translator of the existence and of certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. Each declaration is attached to and valid for one compound statement. A compound statement which includes declarations is called a block.

A program is a self-contained compound statement, i.e. a compound statement which is not contained within another compound statement and which makes no use of other compound statements not contained within it.

In the sequel the syntax and semantics of the language will be given.[4]

---

[4] Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

1.1. Formalism for Syntactic Description

The syntax will be described with the aid of metalinguistic formulae.[5] Their interpretation is best explained by an example

⟨ab⟩ ::= (| [| ⟨ab⟩(| ⟨ab⟩⟨d⟩

Sequences of characters enclosed in the brackets ⟨ ⟩ represent metalinguistic variables whose values are sequences of symbols. The marks ::= and | (the latter with the meaning of **or**) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable ⟨ab⟩. It indicates that ⟨ab⟩ may have the value (or [or that given some legitimate value of ⟨ab⟩, another may be formed by following it with the character (or by following it with some value of the variable ⟨d⟩. If the values of ⟨d⟩ are the decimal digits, some values of ⟨ab⟩ are:

[(((1(37(
(12345(
(((
[86

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e. the sequences of characters appearing within the brackets ⟨ ⟩ as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition:

⟨empty⟩ ::=
(i.e. the null string of symbols).

2. Basic Symbols, Identifiers, Numbers, and Strings. Basic Concepts.

The reference language is built up from the following basic symbols:

⟨basic symbol⟩ ::= ⟨letter⟩|⟨digit⟩|⟨logical value⟩|⟨delimiter⟩

2.1. Letters

⟨letter⟩ ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e. character not coinciding with any digit, logical value or delimiter).

Letters do not have individual meaning. They are used for forming identifiers and strings[6] (cf. sections 2.4. Identifiers, 2.6. Strings).

2.2.1. Digits

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

Digits are used for forming numbers, identifiers, and strings.

2.2.2. Logical Values

⟨logical value⟩ ::= **true**|**false**

The logical values have a fixed obvious meaning.

2.3. Delimiters

⟨delimiter⟩ ::= ⟨operator⟩|⟨separator⟩|⟨bracket⟩|⟨declarator⟩|⟨specificator⟩
⟨operator⟩ ::= ⟨arithmetic operator⟩|⟨relational operator⟩|⟨logical operator⟩|⟨sequential operator⟩
⟨arithmetic operator⟩ ::= $+|-|\times|/|\div|\uparrow$
⟨relational operator⟩ ::= $<|\leqq|=|\geqq|>|\neq$
⟨logical operator⟩ ::= $\equiv|\supset|\vee|\wedge|\neg$
⟨sequential operator⟩ ::= **go to**|**if**|**then**|**else**|**for**|**do**[7]
⟨separator⟩ ::= ,|.|$_{10}$|:|;|:=|␣|**step**|**until**|**while**|**comment**
⟨bracket⟩ ::= (|)|[|]|‘|’|**begin**|**end**
⟨declarator⟩ ::= **own**|**Boolean**|**integer**|**real**|**array**|**switch**|**procedure**
⟨specificator⟩ ::= **string**|**label**|**value**

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols:	*is equivalent with*
; **comment** ⟨any sequence not containing ;⟩;	;
begin comment ⟨any sequence not containing ;⟩;	**begin**
end ⟨any sequence not containing **end** or ; or **else**⟩	**end**

By equivalence is here meant that any of the three symbols shown in the right-hand column may, in any occurrence outside of strings, be replaced by any sequence of symbols of the structure shown in the same line of the left-hand column without any effect on the action of the program.

2.4. Identifiers

2.4.1. Syntax

⟨identifier⟩ ::= ⟨letter⟩|⟨identifier⟩⟨letter⟩|⟨identifier⟩⟨digit⟩

---

[5] Cf. J. W. Backus, The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. ICIP Paris, June 1959.

[6] It should be particularly noted that throughout the reference language boldface is used for defining independent basic symbols (see sections 2.2.2 and 2.3). These are understood to have no relation to the individual letters of which they are composed. Within the present report boldface will be used for no other purpose.

[7] **do** is used in for statements. It has no relation whatsoever to the **do** of the preliminary report, which is not included in ALGOL 60.

2.4.2. Examples

q
Soup
V17a
a34kTMNs
MARILYN

2.4.3. Semantics

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf., however, section 3.2.4. Standard Functions).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7. Quantities, Kinds and Scopes, and section 5. Declarations).

2.5. Numbers

2.5.1. Syntax

⟨unsigned integer⟩ ::= ⟨digit⟩|⟨unsigned integer⟩⟨digit⟩
⟨integer⟩ ::= ⟨unsigned integer⟩|+⟨unsigned integer⟩|
  −⟨unsigned integer⟩
⟨decimal fraction⟩ ::= .⟨unsigned integer⟩
⟨exponent part⟩ ::= $_{10}$⟨integer⟩
⟨decimal number⟩ ::= ⟨unsigned integer⟩|⟨decimal fraction⟩|
  ⟨unsigned integer⟩⟨decimal fraction⟩
⟨unsigned number⟩ ::= ⟨decimal number⟩|⟨exponent part⟩|
  ⟨decimal number⟩⟨exponent part⟩
⟨number⟩ ::= ⟨unsigned number⟩|+⟨unsigned number⟩|
  −⟨unsigned number⟩

2.5.2. Examples

0	-200.084	$-.083_{10}-02$
177	$+07.43_{10}8$	$-_{10}7$
.5384	$9.34_{10}+10$	$_{10}-4$
+0.7300	$2_{10}-4$	$+_{10}+5$

2.5.3. Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4. Types

Integers are of type **integer**. All other numbers are of type **real** (cf. section 5.1. Type Declarations).

2.6. Strings

2.6.1. Syntax

⟨proper string⟩ ::= ⟨any sequence of basic symbols not containing
  ‘ or ’ ⟩|⟨empty⟩
⟨open string⟩ ::= ⟨proper string⟩|‘⟨open string⟩’|
  ⟨open string⟩⟨open string⟩
⟨string⟩ ::= ‘⟨open string⟩’

2.6.2. Examples

‘5k,,−‘[[[‘∧=/:’Tt’’
‘.. This ⌴ is ⌴ a ⌴ ‘string’’

2.6.3. Semantics

In order to enable the language to handle arbitrary sequences of basic symbols the string quotes ‘ and ’ are introduced. The symbol ⌴ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. sections 3.2. Function Designators and 4.7. Procedure Statements).

2.7. Quantities, Kinds and Scopes

The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures.

The scope of a quantity is the set of statements in which the declaration for the identifier associated with that quantity is valid, or, for labels, the set of statements which may have the statement in which the label occurs as their successor.

2.8. Values and Types

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

The various "types" (**integer, real, Boolean**) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational, expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential, operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

⟨expression⟩ ::= ⟨arithmetic expression⟩|⟨Boolean expression⟩|
  ⟨designational expression⟩

3.1. Variables

3.1.1. Syntax

⟨variable identifier⟩ ::= ⟨identifier⟩
⟨simple variable⟩ ::= ⟨variable identifier⟩
⟨subscript expression⟩ ::= ⟨arithmetic expression⟩
⟨subscript list⟩ ::= ⟨subscript expression⟩|⟨subscript list⟩,
  ⟨subscript expression⟩
⟨array identifier⟩ ::= ⟨identifier⟩
⟨subscripted variable⟩ ::= ⟨array identifier⟩[⟨subscript list⟩]
⟨variable⟩ ::= ⟨simple variable⟩|⟨subscripted variable⟩

3.1.2. Examples

epsilon
detA
a17
Q[7,2]
x[sin(n×pi/2),Q[3,n,4]]

3.1.3. Semantics

A variable is a designation given to a single value. This

value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1. Type Declarations) or for the corresponding array identifier (cf. section 5.2. Array Declarations).

3.1.4. Subscripts

3.1.4.1. Subscripted variables designate values which are components of multidimensional arrays (cf. section 5.2. Array Declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3. Arithmetic Expressions).

3.1.4.2. Each subscript position acts like a variable of type integer and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2. Array Declarations).

3.2. Function Designators

3.2.1. Syntax

```
⟨procedure identifier⟩ ::= ⟨identifier⟩
⟨actual parameter⟩ ::= ⟨string⟩|⟨expression⟩|⟨array identifier⟩|
    ⟨switch identifier⟩|⟨procedure identifier⟩
⟨letter string⟩ ::= ⟨letter⟩|⟨letter string⟩⟨letter⟩
⟨parameter delimiter⟩ ::= ,|)⟨letter string⟩:(
⟨actual parameter list⟩ ::= ⟨actual parameter⟩|
    ⟨actual parameter list⟩⟨parameter delimiter⟩
    ⟨actual parameter⟩
⟨actual parameter part⟩ ::= ⟨empty⟩|(⟨actual parameter list⟩)
⟨function designator⟩ ::= ⟨procedure identifier⟩
    ⟨actual parameter part⟩
```

3.2.2. Examples

```
sin(a−b)
J(v+s,n)
R
S(s−5)Temperature:(T)Pressure:(P)
Compile(' := ')Stack:(Q)
```

3.2.3. Semantics

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4. Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7. Procedure Statements. Not every procedure declaration defines the value of a function designator.

3.2.4. Standard functions

Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. It is recommended that this reserved list should contain:

abs(E)	for the modulus (absolute value) of the value of the expression E
sign(E)	for the sign of the value of E(+1 for E>0, 0 for E=0, −1 for E<0)
sqrt(E)	for the square root of the value of E
sin(E)	for the sine of the value of E
cos(E)	for the cosine of the value of E
arctan(E)	for the principal value of the arctangent of the value of E
ln(E)	for the natural logarithm of the value of E
exp(E)	for the exponential function of the value of E (e^E).

These functions are all understood to operate indifferently on arguments both of type **real** and **integer**. They will all yield values of type **real**, except for sign(E) which will have values of type **integer**. In a particular representation these functions may be available without explicit declarations (cf. section 5. Declarations).

3.2.5. Transfer functions

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely

entier(E),

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E.

3.3. Arithmetic Expressions

3.3.1. Syntax

```
⟨adding operator⟩ ::= +|−
⟨multiplying operator⟩ ::= ×|/|÷
⟨primary⟩ ::= ⟨unsigned number⟩|⟨variable⟩|
    ⟨function designator⟩|(⟨arithmetic expression⟩)
⟨factor⟩ ::= ⟨primary⟩|⟨factor⟩↑⟨primary⟩
⟨term⟩ ::= ⟨factor⟩|⟨term⟩⟨multiplying opreator⟩⟨factor⟩
⟨simple arithmetic expression⟩ ::= ⟨term⟩|
    ⟨adding operator⟩⟨term⟩|⟨simple arithmetic expression⟩
    ⟨adding operator⟩⟨term⟩
⟨if clause⟩ ::= if ⟨Boolean expression⟩then
⟨arithmetic expression⟩ ::= ⟨simple arithmetic expression⟩|
    ⟨if clause⟩⟨simple arithmetic expression⟩else
    ⟨arithmetic expression⟩
```

3.3.2. Examples

Primaries:

```
7.394₁₀−8
sum
w[i+2,8]
cos(y+z×3)
(a−3/y+vu↑8)
```

Factors:

```
omega
sum↑cos(y+z×3)
7.394₁₀−8↑w[i+2,8]↑(a−3/y+vu↑8)
```

Terms:

```
U
omega×sum↑cos(y+z×3)/7.394₁₀−8↑w[i+2,8]↑
    (a−3/y+vu−8)
```

Simple arithmetic expression:

```
U−Yu+omega×sum↑cos(y+z×3)/7.394₁₀−8↑w[i+2,8]↑
    (a−3/y+vu↑8)
```

Arithmetic expressions:

```
w×u−Q(S+Cu)↑2
if q>0 then S+3×Q/A else 2×S+3×q
if a<0 then U+V else if a×b>17 then U/V else if
    k≠y then V/U else 0
a×sin(omega×t)
0.57₁₀12×a[N×(N−1)/2, 0]
(A×arctan(y) + Z)↑(7 + Q)
if q then n−1 else n
if a<0 then A/B else if b=0 then B/A else z
```

3.3.3. Semantics

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4. Procedure Declarations) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4. Boolean Expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value **true** is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

else ⟨simple arithmetic expression⟩

is equivalent to the construction:

else if true then ⟨simple arithmetic expression⟩

3.3.4. Operators and types

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types **real** or **integer** (cf. section 5.1. Type Declarations). The meaning of the basic operators and the types of the expressions to whith they lead are given by the following rules:

3.3.4.1. The operators +, −, and × have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be **integer** if both of the operands are of **integer** type, otherwise **real**.

3.3.4.2. The operations ⟨term⟩/⟨factor⟩ and ⟨term⟩÷⟨factor⟩ both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

a/b×7/(p−q)×v/s

means

((((a×(b⁻¹))×7)×((p−q)⁻¹))×v)×(s⁻¹)

The operator / is defined for all four combinations of types **real** and **integer** and will yield results of **real** type in any case. The operator ÷ is defined only for two operands both of type **integer** and will yield a result of type **integer** defined as follows:

a÷b= sign (a/b)×entier(abs(a/b))

(cf. sections 3.2.4 and 3.2.5).

3.3.4.3. The operation ⟨factor⟩↑⟨primary⟩ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example,

2↑n↑k   means $(2^n)^k$

while

2↑(n↑m)   means $2^{(n^m)}$

Writing i for a number of **integer** type, r for a number of **real** type, and a for a number of either **integer** or **real** type, the result is given by the following rules:

```
a↑i  If i>0, a×a×...×a (i times), of the same type as a.
     If i=0, if a≠0, 1, of the same type as a.
             if a=0, undefined.
     If i<0, if a≠0, 1/(a×a×...×a) (the denominator has
                     i factors), of type real.
             if a=0, undefined.
a↑r  If a>0, exp(r×ln(a)), of type real.
     If a=0, if r>0, 0.0, of type real.
             if r≦0, undefined.
     If a<0, always undefined.
```

3.3.5. Precedence of operators

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.3.5.1. According to the syntax given in section 3.3.1 the following rules of precedence hold:

first: ↑
second: ×/÷
third: +−

3.3.5.2. The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6. Arithmetics of **real** quantities

Numbers and variables of type **real** must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation

from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4. Boolean Expressions

3.4.1. Syntax

⟨relational operator⟩ ::= < | ≤ | = | ≥ | > | ≠
⟨relation⟩ ::= ⟨arithmetic expression⟩⟨relational operator⟩
  ⟨arithmetic expression⟩
⟨Boolean primary⟩ ::= ⟨logical value⟩|⟨variable⟩|
  ⟨function designator⟩|⟨relation⟩|(⟨Boolean expression⟩)
⟨Boolean secondary⟩ ::= ⟨Boolean primary⟩|¬⟨Boolean primary⟩
⟨Boolean factor⟩ ::= ⟨Boolean secondary⟩|
  ⟨Boolean factor⟩∧⟨Boolean secondary⟩
⟨Boolean term⟩ ::= ⟨Boolean factor⟩|⟨Boolean term⟩
  ∨⟨Boolean factor⟩
⟨implication⟩ ::= ⟨Boolean term⟩|⟨implication⟩⊃⟨Boolean term⟩
⟨simple Boolean⟩ ::= ⟨implication⟩|
  ⟨simple Boolean⟩≡⟨implication⟩
⟨Boolean expression⟩ ::= ⟨simple Boolean⟩|
  ⟨if clause⟩⟨simple Boolean⟩ **else** ⟨Boolean expression⟩

3.4.2. Examples

x = −2
Y>V ∨ z<q
a+b > −5 ∧ z−d > q↑2
p∧q ∨ x≠y
g≡¬a∧b∧¬c∨d∨e⊃¬f
if k<1 **then** s>w **else** h≤c
if if if a **then** b **else** c **then** d **else** f **then** g **else** h<k

3.4.3. Semantics

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4. Types

Variables and function designators entered as Boolean primaries must be declared **Boolean** (cf. section 5.1. Type Declarations and sections 5.4.4. Values of Function Designators).

3.4.5. The operators

Relations take on the value **true** whenever the corresponding relation is satisfied for the expressions involved, otherwise **false**.

The meaning of the logical operators ¬ (not), ∧ (and), ∨ (or), ⊃ (implies), and ≡ (equivalent), is given by the following function table.

b1	**false**	**false**	**true**	**true**
b2	**false**	**true**	**false**	**true**
¬b1	**true**	**true**	**false**	**false**
b1∧b2	**false**	**false**	**false**	**true**
b1∨b2	**false**	**true**	**true**	**true**
b1⊃b2	**true**	**true**	**false**	**true**
b1≡b2	**true**	**false**	**false**	**true**

3.4.6. Precedence of operators

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1. According to the syntax given in section 3.4.1 the following rules of precedence hold:

first: arithmetic expressions according to section 3.3.5.
second: < ≤ = ≥ > ≠
third: ¬
fourth: ∧
fifth: ∨
sixth: ⊃
seventh: ≡

3.4.6.2. The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.5. Designational Expressions

3.5.1. Syntax

⟨label⟩ ::= ⟨identifier⟩|⟨unsigned integer⟩
⟨switch identifier⟩ ::= ⟨identifier⟩
⟨switch designator⟩ ::= ⟨switch identifier⟩[⟨subscript expression⟩]
⟨simple designational expression⟩ ::= ⟨label⟩|⟨switch designator⟩|
  (⟨designational expression⟩)
⟨designational expression⟩ ::= ⟨simple designational expression⟩|
  ⟨if clause⟩⟨simple designational expression⟩ **else**
  ⟨designational expression⟩

3.5.2. Examples

17
p9
Choose[n−1]
Town[**if** y<0 **then** N **else** N+1]
if Ab<c **then** 17 **else** q[**if** w≦0 **then** 2 **else** n]

3.5.3. Semantics

A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. section 5.3. Switch Declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4. The subscript expression

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1, 2, 2, . . . , n, where n is the number of entries in the switch list.

3.5.5. Unsigned integers as labels

Unsigned integers used as labels have the property that leading zeroes do not affect their meaning, e.g. 00217 denotes the same label as 217.

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1. Compound Statements and Blocks

4.1.1. Syntax

```
⟨unlabelled basic statement⟩ ::= ⟨assignment statement⟩|
    ⟨go to statement⟩|⟨dummy statement⟩|⟨procedure statement⟩
⟨basic statement⟩ ::= ⟨unlabelled basic statement⟩|⟨label⟩:
    ⟨basic statement⟩
⟨unconditional statement⟩ ::= ⟨basic statement⟩|⟨for statement⟩|
    ⟨compound statement⟩|⟨block⟩
⟨statement⟩ ::= ⟨unconditional statement⟩|
    ⟨conditional statement⟩
⟨compound tail⟩ ::= ⟨statement⟩ end |⟨statement⟩   ;
    ⟨compound tail⟩
⟨block head⟩ ::= begin⟨declaration⟩|⟨block head⟩   ;
    ⟨declaration⟩
⟨unlabelled compound⟩ ::= begin ⟨compound tail⟩
⟨unlabelled block⟩ ::= ⟨block head⟩   ;   ⟨compound tail⟩
⟨compound statement⟩ ::= ⟨unlabelled compound⟩|
    ⟨label⟩:⟨compound statement⟩
⟨block⟩::=⟨unlabelled block⟩|⟨label⟩:⟨block⟩
```

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

```
L: L: ... begin S   ;  S   ;  ... S   ;  S end
```

Block:

```
L: L: ... begin D   ;  D   ;  .. D   ;  S   ;  S   ;  ...S   ;
    S end
```

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2. Examples

Basic statements:

```
a := p+q
go to Naples
START: CONTINUE: W := 7.993
```

Compound statement:

```
begin x := 0   ;  for y := 1 step 1 until n do x :=
        x+A[y]   ;
    if x>q then go to STOP else if x>w−2 then
        go to S   ;
    Aw: St: W: = x+bob end
```

Block:

```
Q: begin integer i, k   ;  real w   ;
     for i := 1 step 1 until m do
     for k := i+1 step 1 until m do
     begin w := A[i, k]   ;
             A[i, k] := A[k, i]   ;
             A[k, i] := w end for i and k
   end block Q
```

4.1.3. Semantics

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. section 5. Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be nonlocal to it, i.e. will represent the same entity inside the block and in the level immediately outside it. The exception to this rule is presented by labels, which are local to the block in which they occur.

Since a statement of a block may again itself be a block, the concepts local and non-local to a block must be understood recursively. Thus an identifier, which is non-local to a block A, may or may not be non-local to the block B in which A is one statement.

4.2. Assignment Statements

4.2.1. Syntax

```
⟨left part⟩ ::= ⟨variable⟩ :=
⟨left part list⟩ ::= ⟨left part⟩|⟨left part list⟩⟨left part⟩
⟨assignment statement⟩ ::= ⟨left part list⟩⟨arithmetic expression⟩|
    ⟨left part list⟩⟨Boolean expression⟩
```

4.2.2. Examples

```
s := p[0] := n := n+1+s
n := n+1
A := B/C−v−q×S
s[v,k+2] := 3−arctan(s×zeta)
V := Q>Y∧Z
```

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables. The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.2. The expression of the statement is evaluated.

4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4. Types

All variables of a left part list must be of the same declared type. If the variables are **Boolean**, the expression must likewise be **Boolean**. If the variables are of

real or **integer**, the expression must be arithmetic. If the type of the arithmetic expression differs from that of the variables, appropriate transfer functions are understood to be automatically invoked. For transfer from **real** to **integer** type, the transfer function is understood to yield a result equivalent to

entier(E+0.5)

where E is the value of the expression.

4.3. GO TO STATEMENTS

4.3.1. Syntax

⟨go to statement⟩ ::= **go to** ⟨designational expression⟩

4.3.2. Examples

```
go to S
go to exit [n+1]
go to Town[if y<0 then N else N+1]
go to if Ab<c then 17 else q[if w<0 then 2 else n]
```

4.3.3. Semantics

A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4. Restriction

Since labels are inherently local, no go to statement can lead from outside into a block.

4.3.5. Go to an undefined switch designator

A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4. DUMMY STATEMENTS

4.4.1. Syntax

⟨dummy statement⟩ ::= ⟨empty⟩

4.4.2. Examples

```
L:
begin ... ; John: end
```

4.4.3. Semantics

A dummy statement executes no operation. It may serve to place a label.

4.5. CONDITIONAL STATEMENTS

4.5.1. Syntax

⟨if clause⟩ ::= **if** ⟨Boolean expression⟩ **then**
⟨unconditional statement⟩ ::= ⟨basic statement⟩|⟨for statement⟩|
  ⟨compound statement⟩|⟨block⟩
⟨if statement⟩ ::= ⟨if clause⟩ ⟨unconditional statement⟩|
  ⟨label⟩:⟨if statement⟩
⟨conditional statement⟩ ::= ⟨if statement⟩|⟨if statement⟩ **else**
  ⟨statement⟩

4.5.2. Examples

```
if x>0 then n := n+1
if v>u then V: q: = n+m else go to R
if s<0∨P≦Q then AA: begin if q<v then a := v/s
                else y := 2×a end
              else if v>s then a := v−q else if v>s−1
                then go to S
```

4.5.3. Semantics

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1. If statement. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

if B1 **then** S1 **else if** B2 **then** S2 **else** S3 ; S4

and

if B1 **then** S1 **else if** B2 **then** S2 **else if** B3 **then** S3 ; S4

Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expression of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value **true** is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, i.e. the statement following the complete conditional statement. Thus the effect of the delimiter **else** may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

else ⟨unconditional statement⟩

is equivalent to

else if true then ⟨unconditional statement⟩

If none of the Boolean expressions of the if clauses is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:

```
                 ----------------------------
                 ↑                 ↑        ↓
if B1 then S1 else if B2 then S2 else S3 ; S4
↓               ↑ ↓                ↑
-----------------  ------------------
    B1 false           B2 false
```

4.5.4. Go to into a conditional statement

The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of **else**.

4.6. FOR STATEMENTS

4.6.1. Syntax

⟨for list element⟩ ::= ⟨arithmetic expression⟩|
  ⟨arithmetic expression⟩ **step** ⟨arithmetic expression⟩ **until**
  ⟨arithmetic expression⟩|⟨arithmetic expression⟩ **while**
  ⟨Boolean expression⟩
⟨for list⟩ ::= ⟨for list element⟩|⟨for list⟩, ⟨for list element⟩
⟨for clause⟩ ::= **for** ⟨variable⟩ := ⟨for list⟩ **do**

⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩|
⟨label⟩:⟨for statement⟩

4.6.2. Examples

```
for q := 1 step s until n do A[q] := B[q]
for k := 1, V1×2 while V1<N do
      for j := I+G, L, 1 step 1 until N, C+D do
            A[k,j] := B[k,j]
```

4.6.3. Semantics

A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:

```
                 ---------------------------
                 ↓                         ↑
Initialize ; test ; statement S ; advance ; successor
                 ↓                                  ↑
                  -----------------------------------
                          for list exhausted
```

In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not, the statement following the for clause is executed.

4.6.4. The for list elements

The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1. Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2. Step-until-element. An element for the form A **step** B **until** C, where A, B, and C, are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

```
    V := A  ;
L1: if (V−C)× sign(B)>0 then go to Element exhausted;
    Statement S  ;
    V := V+B  ;
    go to L1  ;
```

where V is the controlled variable of the for clause and Element exhausted points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program.

4.6.4.3. While-element. The execution governed by a for list element of the form E **while** F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

```
L3: V := E  ;
    if ¬F then go to Element exhausted  ;
    Statement S  ;
    go to L3  ;
```

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit

Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will be the same as it was immediately preceding the execution of the go to statement.

If the exit is due to exhaustion of the for list, on the other hand, the value of the controlled variable is undefined after the exit.

4.6.6. Go to leading into a for statement

The effect of a go to statement, outside a for statement, which refers to a label within the for statement, is undefined.

4.7. Procedure Statements

4.7.1. Syntax

⟨actual parameter⟩ ::= ⟨string⟩|⟨expression⟩|⟨array identifier⟩|
⟨switch identifier⟩|⟨procedure identifier⟩
⟨letter string⟩ ::= ⟨letter⟩|⟨letter string⟩⟨letter⟩
⟨parameter delimiter⟩ ::= ,|⟨letter string⟩:(
⟨actual parameter list⟩ ::= ⟨actual parameter⟩|
⟨actual parameter list⟩⟨parameter delimiter⟩
⟨actual parameter⟩
⟨actual parameter part⟩ ::= ⟨empty⟩|
(⟨actual parameter list⟩)
⟨procedure statement⟩ ::= ⟨procedure identifier⟩
⟨actual parameter part⟩

4.7.2. Examples

```
Spur (A)Order: (7)Result to: (V)
Transpose (W,v+1)
Absmax(A,N,M,Yy,I,K)
Innerproduct(A[t,P,u],B[P],10,P,Y)
```

These examples correspond to examples given in section 5.4.2.

4.7.3. Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. section 5.4. Procedure Declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program:

4.7.3.1. Value assignment (call by value)

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and Types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. These formal parameters will subsequently be treated as local to the procedure body.

4.7.3.2. Name replacement (call by name)

Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the

sponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3. Body replacement and execution

Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed.

4.7.4. Actual-formal correspondence

The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.

This poses the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. Strings cannot occur as actual parameters in procedure statements calling procedure declarations having ALGOL 60 statements as their bodies (cf. section 4.7.8).

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4. A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5. Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6. Nonlocal quantities of the body

A procedure statement written outside the scope of any non-local quantity of the procedure body is undefined.

4.7.7. Parameter delimiters

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8. Procedure body expressed in code

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

5. Declarations

Declarations serve to define certain properties of the identifiers of the program. A declaration for an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the **begin**, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through **end**, or by a go to statement) all identifiers which are declared for the block lose their significance again.

A declaration may be marked with the additional declarator **own.** This has the following effect: upon a reentry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax.

⟨declaration⟩ ::= ⟨type declaration⟩|⟨array declaration⟩|
⟨switch declaration⟩|⟨procedure declaration⟩

5.1. TYPE DECLARATIONS

5.1.1. Syntax

⟨type list⟩ ::= ⟨simple variable⟩|
⟨simple variable⟩, ⟨type list⟩
⟨type⟩ ::= **real**|**integer**|**Boolean**
⟨local or own type⟩ ::= ⟨type⟩|**own** ⟨type⟩
⟨type declaration⟩ ::= ⟨local or own type⟩⟨type list⟩

5.1.2. Examples

integer p,q,s
own Boolean Acryl,n

5.1.3. Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values **true** and **false**.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of **own**, see the fourth paragraph of section 5 above.

5.2. Array Declarations

5.2.1. Syntax

```
⟨lower bound⟩ ::= ⟨arithmetic expression⟩
⟨upper bound⟩ ::= ⟨arithmetic expression⟩
⟨bound pair⟩ ::= ⟨lower bound⟩:⟨upper bound⟩
⟨bound pair list⟩ ::= ⟨bound pair⟩|⟨bound pair list⟩,⟨bound pair⟩
⟨array segment⟩ ::= ⟨array identifier⟩[⟨bound pair list⟩]|
    ⟨array identifier⟩,⟨array segment⟩
⟨array list⟩ ::= ⟨array segment⟩|⟨array list⟩,⟨array segment⟩
⟨array declaration⟩ ::= array ⟨array list⟩|⟨local or own type⟩
    array ⟨array list⟩
```

5.2.2. Examples

array a, b, c[7:n,2:m], s[−2:10]
own integer array A[**if** c<0 **then** 2 **else** 1:20]
real array q[−7:−1]

5.2.3. Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

5.2.3.1. Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter: The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3. Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type **real** is understood.

5.2.4. Lower upper bound expressions

5.2.4.1. The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2).

5.2.4.2. The expressions can only depend on variables and procedures which are non-local to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4. The expressions will be evaluated once at each entrance into the block.

5.2.5. The identity of subscripted variables

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared **own** the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3. Switch Declarations

5.3.1. Syntax

```
⟨switch list⟩ ::= ⟨designational expression⟩|
    ⟨switch list⟩,⟨designational expression⟩
⟨switch declaration⟩ ::= switch ⟨switch identifier⟩:=⟨switch list⟩
```

5.3.2. Examples

switch S := S1,S2,Q[m], **if** v>−5 **then** S3 **else** S4
switch Q := p1,w

5.3.3. Semantics

A switch declaration defines the values corresponding to a switch identifier. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1, 2, ..., obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5. Designational Expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4. Evaluation of expressions in the switch list

An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5. Influence of scopes.

Any reference to the value of a switch designator from outside the scope of any quantity entering into the designational expression for this particular value is undefined.

5.4. Procedure Declarations

5.4.1. Syntax

```
⟨formal parameter⟩ ::= ⟨identifier⟩
⟨formal parameter list⟩ ::= ⟨formal parameter⟩|
    ⟨formal parameter list⟩⟨parameter delimiter⟩
    ⟨formal parameter⟩
⟨formal parameter part⟩ ::= ⟨empty⟩|(⟨formal parameter list⟩)
⟨identifier list⟩ ::= ⟨identifier⟩|⟨identifier list⟩,⟨identifier⟩
⟨value part⟩ ::=value⟨identifier list⟩ ; |⟨empty⟩
⟨specifier⟩ ::= string|⟨type⟩|array|⟨type⟩array|label|switch|
    procedure|⟨type⟩procedure
⟨specification part⟩ ::= ⟨empty⟩|⟨specifier⟩⟨identifier list⟩;|
    ⟨specification part⟩⟨specifier⟩⟨identifier list⟩ ;
```

```
<procedure heading> ::= <procedure identifier>
    <formal parameter part> ; <value part><specification part>
<procedure body> ::= <statement>|<code>
<procedure declaration> ::=
    procedure <procedure heading><procedure body>|
    <type> procedure <procedure heading><procedure body>
```

5.4.2. Examples (see also the examples at the end of the report).

```
procedure Spur(a)Order:(n)Result:(s) ; value n ;
array a ; integer n ; real s ;
begin integer k ;
s := 0 ;
for k := 1 step 1 until n do s := s + a[k,k]
end

procedure Transpose(a)Order:(n) ; value n ;
array a ; integer n ;
begin real w ; integer i, k ;
for i := 1 step 1 until n do
      for k := 1+i step 1 until n do
      begin w := a[i,k] ;
            a[i,k] := a[k,i] ;
            a[k,i] := w
      end
end Transpose

integer procedure Step (u) ; real u ;
Step := if 0≦u∧u≦1 then 1 else 0

procedure Absmax(a)size:(n,m)Result:(y)Subscripts:
    (i,k) ;
comment The absolute greatest element of the matrix a,
    of size n by m is transferred to y, and the sub-
    scripts of this element to i and k ;
array a ; integer n, m, i, k ; real y ;
begin integer p, q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until
    m do
if abs(a[p,q])>y then begin y:=abs(a[p,q]) ; i:=p ;
    k:=q
end end Absmax

procedure  Innerproduct(a,b)Order:(k,p)Result:(y) ;
    value k ;
integer k,p ; real y,a,b, ;
begin real s ;
s:= 0 ;
for p := 1 step 1 until k do s :=s+a×b ;
y := s
end Innerproduct
```

5.4.3. Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. section 3.2. Function Designators and section 4.7. Procedure Statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or non-local to the body depending on whether they are declared within the body or not. Those of them which are nonlocal to the body may well be local to the block in the head of which the procedure declaration appears.

5.4.4. Values of function designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur an assignment of a value to the procedure identifier, and in addition the type of this value must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration.

Any other occurrence of the procedure identifier within the procedure body denotes activation of the procedure.

5.4.5. Specifications

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once and formal parameters called by name (cf. section 4.7.3.2) may be omitted altogether.

5.4.6. Code as procedure body

It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language.

Examples of Procedure Declarations:

EXAMPLE 1.

```
procedure euler (fct, sum, eps, tim) ; value eps, tim ;
integer tim ; real procedure fct ; real sum, eps ;
comment euler computes the sum of fct(i) for i from zero up to
infinity by means of a suitably refined euler transformation. The
summation is stopped as soon as tim times in succession the abso-
lute value of the terms of the transformed series are found to be
less than eps. Hence, one should provide a function fct with one
integer argument, an upper bound eps, and an integer tim. The
output is the sum sum.  euler is particularly efficient in the case
of a slowly convergent or divergent alternating series ;
begin integer i, k, n, t ; array m[0:15] ; real mn, mp, ds ;
i := n := t := 0 ; m[0] := fct(0) ; sum := m[0]/2 ;
nextterm: i := i+1 ; mn := fct(i) ;
          for k := 0 step 1 until n do
               begin mp := (mn+m[k])/2 ; m[k] := mn ;
                 mn := mp end means ;
          if (abs(mn)<abs(m[n]))∧(n<15) then
               begin ds := mn/2 ; n := n+1 ; m[n] :=
                 mn end accept
          else ds := mn ;
          sum := sum + ds ;
          if abs(ds)<eps then t := t+1 else t := 0 ;
          if t<tim then go to nextterm
end euler
```

EXAMPLE 2.[5]

```
procedure RK(x,y,n,FKT,eps,eta,xE,yE,fi)   ; value x,y   ;
integer n   ; Boolean fi   ; real x,eps,eta,xE   ; array
y,yE   ; procedure FKT   ;
comment : RK integrates the system yk'=fk(x,y1,y2,... ,yn)
(k=1,2,... ,n) of differential equations with the method of Runge-
Kutta with automatic search for appropriate length of integration
step. Parameters are: The initial values x and y[k] for x and the un-
known functions yk(x). The order n of the system. The procedure
FKT(x,y,n,z) which represents the system to be integrated, i.e.
the set of functions fk . The tolerance values eps and eta which
govern the accuracy of the numerical integration. The end of the
integration interval xE. The output parameter yE which repre-
sents the solution at x=xE. The Boolean variable fi, which must
always be given the value true for an isolated or first entry into
RK. If however the functions y must be available at several mesh-
points x0 , x1 , ... , xn , then the procedure must be called repeat-
edly (with x=xk ,  xE=xk+1,  for k=0, 1, ... , n-1) and then the
later calls may occur with fi=false which saves computing time.
The input parameters of FKT must be x,y,n, the output parameter
z represents the set of derivatives z[k]=fk(x,y[1], y[2], ... , y[n])
for x and the actual y's. A procedure comp enters as a non-local
identifier   ;
begin
  array z,y1,y2,y3[1:n]   ; real x1,x2,x3,H   ; Boolean out   ;
  integer k,j   ; own real s,Hs   ;
  procedure RK1ST(x,y,h,xe,ye)   ; real x,h,xe   ; array
          y,ye   ;
    comment : RK1ST integrates one single RUNGE-KUTTA
          with initial values x,y[k] which yields the output
          parameters xe=x+h and ye[k], the latter being the
          solution at xe. Important: the parameters n, FKT,
          enter RK1ST as nonlocal entities   ;
  begin
    array w[1:n], a[1:5]   ; integer k,j   ;
    a[1] := a[2] := a[5] := h/2   ; a[3] := a[4] := h   ;
    xe := x   ;
    for k := 1 step 1 until n do ye[k] := w[k] := y[k]   ;
    for j := 1 step 1 until 4 do
    begin
      FKT(xe,w,n,z)   ;
      xe := x+a[j]   ;
      for k := 1 step 1 until n do
      begin
        w[k] := y[k]+a[j]×z[k]   ;
        ye[k] := ye[k] + a[j+1]×z[k]/3
      end k
    end j
  end RK1ST   ;
Begin of program:
    if fi then begin H := xE-x   ; s := 0 end else H := Hs   ;
    out := false   ;
AA: if (x+2.01×H-xE>0)≡(H>0) then
    begin Hs := H   ; out := true   ; H := (xE-x)/2
      end if   ;
    RK1ST (x,y,2×H,x1,y1)   ;
BB: RK1ST   (x,y,H,x2,y2)   ; RK1ST(x2,y2,H,x3,y3)   ;
    for k := 1 step 1 until n do
          if comp(y1[k],y3[k],eta)>eps then go to CC   ;
    comment : comp(a,b,c) is a function designator, the value
      of which is the absolute value of the difference of the
      mantissae of a and b, after the exponents of these
      quantities have been made equal to the largest of the ex-
      ponents of the originally given parameters a,b,c   ;
    x := x3   ; if out then go to DD   ;
    for k := 1 step 1 until n do y[k] := y3[k]   ;
    if s=5 then begin s := 0   ; H := 2×H end if   ;
    s := s+1   ; go to AA   ;
CC: H := 0.5×H   ; out := false   ; x1 := x2   ;
    for k := 1 step 1 until n do y1[k] := y2[k]   ;
    go to BB   ;
DD: for k := 1 step 1 until n do yE[k] := y3[k]
end RK
```

[5] This RK-program contains some new ideas which are related to ideas of S. GILL, A process for the step-by-step integration of differential equations in an automatic computing machine, *Proc. Camb. Phil. Soc. Vol. 47* (1951) p. 96; and E. FRÖBERG, On the solution of ordinary differential equations with digital computing machines, *Fysiograf. Sällsk. Lund, Förhd. 20* Nr. 11 (1950) p. 136–152. It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

All references are given through section numbers. The references are given in three groups:

def Following the abbreviation "def", reference to the syntactic definition (if any) is given.

synt Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.

text Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words have been collected at the beginning. The examples have been ignored in compiling the index.

+, see: plus
−, see: minus
×, see: multiply
/, ÷, see: divide
↑, see: exponentiation
<, ≦, =, ≧, >, ≠, see: ⟨relational operator⟩
≡, ⊃, ∨, ∧, ¬, see: ⟨logical operator⟩
,, see: comma
., see: decimal point
₁₀, see: ten
:, see: colon
;, see: semicolon
:=, see: colon equal
␣, see: space
( ), see: parentheses
[ ], see: subscript bracket
' ', see: string quote

⟨actual parameter⟩, def 3.2.1, 4.7.1
⟨actual parameter list⟩, def 3.2.1, 4.7.1
⟨actual parameter part⟩, def 3.2.1, 4.7.1
⟨adding operator⟩, def 3.3.1
alphabet, text 2.1
arithmetic, text 3.3.6
⟨arithmetic expression⟩, def 3.3.1 synt 3, 3.1.1, 3.3.1, 3.4.1, 4.2.1, 4.6.1, 5.2.1 text 3.3.3
⟨arithmetic operator⟩, def 2.3 text 3.3.4
array, synt 2.3, 5.2.1, 5.4.1
array, text 3.1.4.1
⟨array declaration⟩, def 5.2.1 synt 5 text 5.2.3
⟨array identifier⟩, def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
⟨array list⟩, def 5.2.1
⟨array segment⟩, def 5.2.1
⟨assignment statement⟩, def 4.2.1 synt 4.1.1 text 1, 4.2.3

⟨basic statement⟩, def 4.1.1 synt 4.5.1
⟨basic symbol⟩, def 2
begin, synt 2.3, 4.1.1
⟨block⟩, def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
⟨block head⟩, def 4.1.1
Boolean, synt 2.3, 5.1.1 text 5.1.3
⟨Boolean expression⟩, def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1 text 3.4.3
⟨Boolean factor⟩, def 3.4.1
⟨Boolean primary⟩, def 3.4.1
⟨Boolean secondary⟩, def 3.4.1
⟨Boolean term⟩, def 3.4.1
⟨bound pair⟩, def 5.2.1
⟨bound pair list⟩, def 5.2.1
⟨bracket⟩, def 2.3

⟨code⟩, synt 5.4.1 text 4.7.8, 5.4.6
colon :, synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1
comma ,, synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1
comment, synt 2.3
comment convention, text 2.3
⟨compound statement⟩, def 4.1.1 synt 4.5.1 text 1
⟨compound tail⟩, def 4.1.1
⟨conditional statement⟩, def 4.5.1 synt 4.1.1 text 4.5.3

⟨decimal fraction⟩, def 2.5.1
⟨decimal number⟩, def 2.5.1 text 2.5.3
decimal point ., synt 2.3, 2.5.1
⟨declaration⟩, def 5 synt 4.1.1 text 1, 5 (complete section)
⟨declarator⟩, def 2.3
⟨delimiter⟩, def 2.3 synt 2
⟨designational expression⟩, def 3.5.1 synt 3, 4.3.1, 5.3.1 text 3.5.3
⟨digit⟩, def 2.2.1 synt 2, 2.4.1, 2.5.1
dimension, text 5.2.3.2
divide / ÷, synt 2.3, 3.3.1 text 3.3.4.2
do, synt 2.3, 4.6.1
⟨dummy statement⟩, def 4.4.1 synt 4.1.1 text 4.4.3

else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.5.1 text 4.5.3.2
⟨empty⟩, def 1.1 synt 2.6.1, 3.2.1, 4.4.1, 4.7.1, 5.4.1
end, synt 2.3, 4.1.1
entier, text 3.2.5
exponentiation ↑, synt 2.3, 3.3.1 text 3.3.4.3
⟨exponent part⟩, def 2.5.1 text 2.5.3
⟨expression⟩, def 3 synt 3.2.1, 4.7.1 text 3 (complete section)

⟨factor⟩, def 3.3.1
false, synt 2.2.2
for, synt 2.3, 4.6.1
⟨for clause⟩, def 4.6.1 text 4.6.3
⟨for list⟩, def 4.6.1 text 4.6.4
⟨for list element⟩, def 4.6.1 text 4.6.4.1, 4.6.4.2, 4.6.4.3
⟨formal parameter⟩, def 5.4.1 text 5.4.3
⟨formal parameter list⟩, def 5.4.1
⟨formal parameter part⟩, def 5.4.1
⟨for statement⟩, def 4.6.1 synt 4.1.1, 4.5.1 text 4.6 (complete section)
⟨function designator⟩, def 3.2.1 synt 3.3.1, 3.4.1 text 3.2.3, 5.4.4

go to, synt 2.3, 4.3.1
⟨go to statement⟩, def 4.3.1 synt 4.1.1 text 4.3.3

⟨identifier⟩, def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1 text 2.4.3
⟨identifier list⟩, def 5.4.1
if, synt 2.3, 3.3.1, 4.5.1
⟨if clause⟩, def 3.3.1, 4.5.1 synt 3.4.1, 3.5.1 text 3.3.3, 4.5.3.2
⟨if statement⟩, def 4.5.1 text 4.5.3.1
⟨implication⟩, def 3.4.1
integer, synt 2.3, 5.1.1 text 5.1.3
⟨integer⟩, def 2.5.1 text 2.5.4

label, synt 2.3, 5.4.1
⟨label⟩, def 3.5.1 synt 4.1.1, 4.5.1, 4.6.1 text 1 4.1.3
⟨left part⟩, def 4.2.1
⟨left part list⟩, def 4.2.1
⟨letter⟩, def 2.1 synt 2, 2.4.1, 3.2.1, 4.7.1
⟨letter string⟩, def 3.2.1, 4.7.1
local, text 4.1.3
⟨local or own type⟩, def 5.1.1 synt 5.2.1
⟨logical operator⟩, def 2.3 synt 3.4.1 text 3.4.5
⟨logical value⟩, def 2.2.2 synt 2, 3.4.1
⟨lower bound⟩, def 5.2.1 text 5.2.4

minus −, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
multiply ×, synt 2.3, 3.3.1 text 3.3.4.1
⟨multiplying operator⟩, def 3.3.1

nonlocal, text 4.1.3
⟨number⟩, def 2.5.1 text 2.5.3, 2.5.4

⟨open string⟩, def 2.6.1
⟨operator⟩, def 2.3
own, synt 2.3, 5.1.1 text 5, 5.2.5

⟨parameter delimiter⟩, def 3.2.1, 4.7.1 synt 5.4.1 text 4.7.7
parentheses ( ), synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1, 4.7.1, 5.4.1 text 3.3.5.2
plus +, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
⟨primary⟩, def 3.3.1
procedure, synt 2.3, 5.4.1
⟨procedure body⟩, def 5.4.1
⟨procedure declaration⟩, def 5.4.1 synt 5 text 5.4.3
⟨procedure heading⟩, def 5.4.1 text 5.4.3
⟨procedure identifier⟩ def 3.2.1 synt 3.2.1, 4.7.1, 5.4.1 text 4.7.5.4
⟨procedure statement⟩, def 4.7.1 synt 4.1.1 text 4.7.3
program, text 1
⟨proper string⟩, def 2.6.1

quantity, text 2.7

real, synt 2.3, 5.1.1 text 5.1.3
⟨relation⟩, def 3.4.1 text 3.4.5
⟨relational operator⟩, def 2.3, 3.4.1

scope, text 2.7
semicolon ;, synt 2.3, 4.1.1, 5.4.1
⟨separator⟩, def 2.3
⟨sequential operator⟩, def 2.3
⟨simple arithmetic expression⟩, def 3.3.1 text 3.3.3
⟨simple Boolean⟩, def 3.4.1
⟨simple designational expression⟩, def 3.5.1

⟨simple variable⟩, def 3.1.1 synt 5.1.1 text 2.4.3
space ␣, synt 2.3 text 2.3, 2.6.3
⟨specification part⟩, def 5.4.1 text 5.4.5
⟨specificator⟩, def 2.3
⟨specifier⟩, def 5.4.1
standard function, text 3.2.4, 3.2.5
⟨statement⟩, def 4.1.1, synt 4.5.1, 4.6.1, 5.4.1 text 4 (complete section)
statement bracket, see: **begin** **end**
step, synt 2.3, 4.6.1 text 4.6.4.2
string, synt 2.3, 5.4.1
⟨string⟩, def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
string quotes ‘ ’, synt 2.3, 2.6.1, text 2.6.3
subscript, text 3.1.4.1
subscript bound, text 5.2.3.1
subscript bracket [ ], synt 2.3, 3.1.1, 3.5.1, 5.2.1
⟨subscripted variable⟩, def 3.1.1 text 3.1.4.1
⟨subscript expression⟩, def 3.1.1 synt 3.5.1
⟨subscript list⟩, def 3.1.1
successor, text 4
switch, synt 2.3, 5.3.1, 5.4.1
⟨switch declaration⟩, def 5.3.1 synt 5 text 5.3.3
⟨switch designator⟩, def 3.5.1 text 3.5.3
⟨switch identifier⟩, def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
⟨switch list⟩, def 5.3.1

⟨term⟩, def 3.3.1
ten $_{10}$, synt 2.3, 2.5.1
then, synt 2.3, 3.3.1, 4.5.1
transfer function, text 3.2.5
true, synt 2.2.2
⟨type⟩, def 5.1.1 synt 5.4.1 text 2.8
⟨type declaration⟩, def 5.1.1 synt 5 text 5.1.3
⟨type list⟩, def 5.1.1

⟨unconditional statement⟩, def 4.1.1, 4.5.1
⟨unlabelled basic statement⟩, def 4.1.1
⟨unlabelled block⟩, def 4.1.1
⟨unlabelled compound⟩, def 4.1.1
⟨unsigned integer⟩, def 2.5.1, 3.5.1
⟨unsigned number⟩, def 2.5.1 synt 3.3.1
until, synt 2.3, 4.6.1 text 4.6.4.2
⟨upper bound⟩, def 5.2.1 text 5.2.4

value, synt 2.3, 5.4.1
value, text 2.8, 3.3.3
⟨value part⟩, def 5.4.1 text 4.7.3.1
⟨variable⟩, def 3.1.1 synt 3.3.1, 3.4.1, 4.2.1, 4.6.1 text 3.1.3
⟨variable identifier⟩, def 3.1.1

while, synt 2.3, 4.6.1 text 4 6.4.3

END OF THE REPORT

[NOTE: Reproduction of this Report for any purpose is explicitly permitted; reference should be made to this issue of the *Communications* as the source. Orders for reprints of this Report may be placed until June 30, 1960 directly with the printer of *Communications*.]